A Practical Testing Framework for Isolating Hardware Timing Channels

Jason Oberg*, Sarah Meiklejohn*, Timothy Sherwood† and Ryan Kastner*
*Computer Science and Engineering, University of California, San Diego
{jkoberg,smeiklej,kastner}@cs.ucsd.edu
†Computer Science, University of California, Santa Barbara
sherwood@cs.ucsb.edu

***Abstract*—This work identifies a new formal basis for hardware information flow security by providing a method to separate timing flows from other flows of information. By developing a framework for identifying these different classes of information flow at the gate-level, one can either confirm or rule out the existence of such flows in a provable manner. To demonstrate the effectiveness of our presented model, we discuss its usage on a practical example: a CPU cache in a MIPS processor written in Verilog HDL and simulated in a scenario which accurately models previous cache-timing attacks. We demonstrate how our framework can be used to isolate the timing channel used in these attacks.**

I. Introduction

Recent work on hardware and embedded system security analysis has shown that it is now possible to constrain the flow of information in both provable and useful ways. When these systems control our pacemakers, our automobiles, our insulin pumps, and our commercial airlines, engineers may be asked to go to more extreme measures to ensure the saftey and security of their computations. Information flow tracking at the gate level [1] can capture many potential issues in a system, and useful designs can be created with very strong security properties. Unfortunately strong properties, such as non-interference, come at a cost.

To ensure that two parts of a system are non-interfering, it has to be shown that one part can have *absolutely* no effect on the other. There are many sources of interference which range from directly modifying functionality to simply changing the timing of events. For example, two parts of a system that opportunistically share a cache might never interfere with one another's data, but are not non-interfering because they can affect the time it takes for them to perform their duties. These timing variations are not always benign, and in fact, as we have seen in prior work, they can lead to leaked keys and other serious problems [2]. Unfortunately, fixing these timing leaks does not come without a cost. And in some cases, a designer may not wish to sacrifice peformance in order to gain security, i.e., there are cases where affecting only the *time* a system takes to perform its work may not be an important consideration. Unfortunately existing techniques (including gate-level information flow tracking (GLIFT) [1]) make no separation between timing and function for the purposes of examining system behavior, thus all of the provable work at the hardware level breaks down.

Even though these timing channels may or may not be in the system's threat model, security-conscious hardware designers must have methods to separate these timing-only flows from more direct information flows to be able to make informed decisions about the system's security. To help make this separation possible, we present a formal technique that can identify the existence of functional information and, when used in conjunction with previous information flow tracking work in hardware, isolate timing information.

Our technique is quite different from that of past work. The most prominent work in identifying timing channels is by Kemmerer et al. [3], who use an informal shared-resource matrix to pin-point where the potential timing channels may appear. This method helps at the higher computing abstractions, but becomes difficult to use when designs become more embedded and application-specific. Our formal method focuses on the hardware design itself so that the system can be built with a secure root-of-trust, thus providing security assurance for the higher abstractions. More *ad-hoc* approaches [4] focus on introducing random noise into the system to make extracting information stochastically difficult. These methods make a timing channel harder to exploit (lower signal-to-noise ratio), but fail at identifying if a channel is timing-based, as we do in this work. The information flow tracking strategies which target hardware description languages [5], [6] themselves work well at preventing timing channels but are quite different than our work. These languages force the designer to rewrite their code in the new language. On the other hand, the formal method we present here can be directly applied to existing IP cores without requiring code rewriting.

Constructing a formal method to separate timing and functional flows requires some assumptions in order for its usage to be practical given the resources system and hardware designers have. Ideally, a designer would be given a system with some secret inputs and determine with complete confidence how the inputs flow (if at all) to unprivileged outputs through its functional behavior or its timing. Unfortunately, such a

978-3-9815370-0-0/DATE13/©2013 EDAA

guarantee is likely to be too strong to prove in practice, especially with the growing complexity of embedded systems. Rather than demonstrating provable non-interference, this work therefore strives to provide a practical testing framework that fits well with existing testing techniques and tools. By relaxing such strong guarantees, our framework can be easily used by engineers and system designers to test for and separate functional and timing flows.

To show the practicality of our framework, we explore in Section III a common shared resource which is at the heart of interference in modern systems: the CPU cache. As previously mentioned, the cache is a common vulnerability in systems, as it is typically susceptible to leaking secret information through time. We show how the information leak in the cache is directly from a timing channel. For this example, we do not make claims about complete information security, but rather higher assurance in identifying the presence of functional information and separating it from timing channels.

II. Isolating Timing Channels

As discussed in past work [7], GLIFT allows system designers to determine if any information flows exist within their systems; we clarify here that we use information flow to mean a logical flow, as we consider other types of flows (e.g., physical phenomena such as electromagnetic radiation or power fluctuations) as out of the scope of this work. Logical flows can be further broken down into two types: *functional* flows and *timing* flows. Intuitively, a functional flow exists for a given set of inputs to a system if their values affect the values output by the system (for example, changing the value of a will affect the output of the function $f(a, b) := a + b$), while a timing flow exists if information about the input can be learned from the latency of the execution. While GLIFT will tell the designer only if any such flow exists, we describe in this section how to determine whether or not the system contains specifically functional flows. Used in conjunction with GLIFT as shown in Figure 1, this technique allows us to also determine if timing flows (and therefore channels) exist. If GLIFT determines that a flow does exist but we can demonstrate that no functional flow exists, then we know that a timing flow must exist. What is left open, however, is the case in which GLIFT determines that a flow exists but we determine that a functional flow does exist; in this case, we are unable to determine if a timing flow exists as well.

A. Finding functional flows

Before we describe how to determine whether or not functional flows exist, we must first define functional flows and related notions formally. We start with the notion of time; as we are working at the gate level, the only notion of time that we consider is the system clock.

Definition 1. *The* clock *is a function with no inputs that outputs values of the form $b \in \{0, 1\}$. A* clock tick *is the event in which the output of the clock changes. Finally, a* time *t is the number of clock ticks that have occurred, and T is the set containing all possible values of t.*

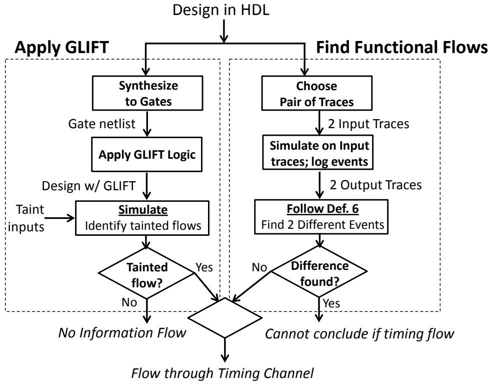

Fig. 1. How our method can be used with GLIFT to isolate timing channels. If GLIFT says theres a flow and we do not find a functional flow, we know there exists a timing channel. If we find a functional flow (GLIFT will also indicate a flow) we cannot conclude the existence of a timing channel.

Our formal definition of time captures what we intuitively expect: some stateless hardware component will output a stream of ticks, and a separate stateful component will measure the number of ticks and use this to keep track of time. By keeping track of time, we can define an *event* as a given value at a certain point in time.

Definition 2. (from [8]) *For a set Y, a* discrete event *is the pair $e := (y, t)$ for $y \in Y$ and $t \in T$ (where we recall T is the set of all possible time values). We also define functions that recover the value and time components of an event as $\mathsf{val}(e) = y$ and $\mathsf{time}(e) = t$ respectively.*

To keep track of how values change over time, we can also define a sequence of events as a *trace*.

Definition 3. *For a value $n \in \mathbb{N}$ and a set Y, a* trace *$A(Y, n)$ is a sequence of discrete events $\{e_i = (y_i, t_i)\}_{i=1}^{n}$ that is ordered by time; i.e., $\mathsf{time}(e_i) < \mathsf{time}(e_{i+1})$ for all i, $1 \leq i < n$, and such that $\mathsf{val}(e_i) \in Y$, $\mathsf{time}(e_i) \in T$ for all i, $1 \leq i \leq n$. When the values of Y and n are clear, we omit them and refer to the trace simply as A.*

The way in which we have currently defined an event is quite broad: any value at any time can be considered an event. In many cases, however, events in this trace may be redundant, as the system might output the same value for many clock ticks while performing some computation. In this case, we would be interested not in the entire progression of events, but only in the case when the value of the output changes. To capture this, we define the *distinct* trace.

Definition 4. *For a trace $A(Y, n)$, the* distinct trace *of A is the largest subsequence $d(A) \subseteq A(Y, n)$ such that for all $e_{i-1}, e_i \in d(A)$ it holds that $\mathsf{val}(e_i) \neq \mathsf{val}(e_{i-1})$.*

Constructing the distinct trace $d(A)$ of A is quite simple: first, include the first element of A in $d(A)$. Next, for each subsequent event e, check whether the last event e' in $d(A)$ is such that $\mathsf{val}(e') = \mathsf{val}(e)$; if this holds, then skip e (i.e., do not include it) and if it doesn't then add e to $d(A)$.

With these definitions in hand, we can now attempt to model some system S that takes as input a value x in some set X and returns a value y in some set Y. To be fully general and consider systems that take input and output vectors rather than single elements, we assume that $X = X_1 \times \ldots \times X_n$ and that $Y = Y_1 \times \ldots \times Y_m$ for some $m, n \geq 1$, which means that an input x looks like $x = (x_1, \ldots, x_n)$ and an output y looks like $y = (y_1, \ldots, y_m)$. To furthermore acknowledge that the system is not static and thus both the inputs and outputs might change over time, we instead provide as input a trace $A(X, k)$ for some value k, and assume our output is a trace $A(Y, k)$.

We are now ready to begin discussing functional flows. Recall first our intuition: a functional flow exists for a set of inputs I to the system S if their values affect the value of the output. One natural way to then test whether or not the value of these inputs affects the value of the output is to change their value and see if the value of the output changes; concretely, this would mean running S on two different traces, in which the values of these inputs are different. In order to isolate just this set I, however, it is necessary to keep the value of the other inputs the same. To ensure that this happens, we define what it means for two traces to be *value preserving*.

Definition 5. *For a set of inputs $\{x_i\}$ for $i \in I$ and two traces $A(X, k) = (e_1, \ldots, e_k)$ and $A(X, k)' = (e'_1, \ldots, e'_k)$, we say the traces are* value preserving *with respect to I if for all $e_j \in A$ and $e'_j \in A'$ it is the case that $\mathsf{time}(e_j) = \mathsf{time}(e'_j)$, and if $\mathsf{val}(e_j) = (a_1, \ldots, a_n)$ and $\mathsf{val}(e'_j) = (a'_1, \ldots, a'_n)$, then $a_i = a'_i$ for all $i \notin I$.*

If two traces are value preserving, then by this definition we know that the only difference between them is the value of the inputs $\{x_i\}$, which is exactly what we need to test for functional flows. We start by defining a relatively weak definition for a functional flow where $S(A)$ denotes S executing on input trace A:

Definition 6 (Functional flow)**.** *For a deterministic system implementation S with input space X and output space Y, we say that a* functional flow *exists with respect to a set of inputs $\{x_i\}_{i \in I}$ and input traces $A(X, k)$ and $A(X, k)'$ that are value preserving with respect to I if for $B := S(A)$ and $B' := S(A')$ it is the case that there exists events $e_j \in d(B)$ and $e'_j \in d(B')$ such that $\mathsf{val}(e_j) \neq \mathsf{val}(e'_j)$.*

Having no functional flow says that, given the output B, by observing B' as well, we do not learn any additional functional information about the inputs $\{x_i\}$ beyond what we learned just from seeing B. It does not, however, strictly guarantee that a functional flow does not exist, as it might exist in the context of two other traces beyond the ones we consider; a definition that would truly rule out functional flows would therefore say that a functional flow does not exist if two such input traces do not exist (due to space constraints we omit this formal definition here). While our definition therefore provides weaker guarantees on the existence of a functional flow, it allows for the most efficient testing: we need to pick only two traces, rather than attempt to enumerate over the entire space. While this does not imply the complete lack of any functional flow, running this procedure with more pairs of traces would only strengthen the evidence.

III. Cache Timing Channel

Recent work has shown CPU caches to be one of the biggest sources of hardware timing channels in modern processors [2]. Many data encryption algorithms, such as the advanced encryption standard (AES), use look-up tables based on the value of the secret key. Since a look-up table will return a value in an amount of time that is directly correlated with whether or not the value is already cached, observing the timing of interactions with the look-up table could produce valuable information about the secret key. In previous work, this vulnerability has been used to completely extract the secret key of AES using three different types of attacks. In this work, we chose to look at an access-driven attack used by Osvik et. al [2] — although the methods presented here can also be applied to the other classes of attacks — as it is the easiest for us to demonstrate given our current test setup.

A. Identifying the Cache Attack as a Timing Channel

In the access-driven cache timing attack on AES used by Osvik et al. [2] (specifically their Prime+Probe variant), a malicious process (M) first fills the contents of the cache by reading a fixed block of data. Next, a secret process (V) uses an unknown key to perform encryption. Finally, M reads the same block of data and observes which cache lines were evicted based on the latency of its memory accesses. Since the key is used to index into look-up tables, the malicious process can correlate fast accesses with the value of the secret key. This cache attack is clearly a type of timing attack, as it critically relies on the timing information available to M. In this section, we demonstrate this more formally by using GLIFT and our model from Section II to prove that these flows are temporal.

We first designed a complete MIPS-based processor written in Verilog. The processor is capable of running several of the SPEC 2006 benchmarks including `mcf`, `specrand`, and `bzip2`, in addition to two security benchmarks: `sha` and `aes`, all of which are executed on the processor by being simulated in ModelSim SE 10.0a. All benchmarks are cross-compiled to MIPS assembly using the SESC gcc compiler; the binary is then loaded into instruction memory using a Verilog testbench. The architecture of the processor consists of a 5-stage pipeline and 32 entry direct mapped cache (e.g. 1-way), although we note that our analysis applies directly to a cache with greater associativity. We chose to use a small cache to speed up the simulation and reduce synthesis time.

Since our particular region of interest is the cache, we focus our analysis directly on this subsystem. To do so, we apply GLIFT logic to the cache system as described in past work [7]. Specifically, we remove the hardware modules associated with the cache (cache control logic and the memory itself) and synthesize them to logic gates and flip-flops using Synopsys' Design Compiler targeting its `and_or.db` library. We then use our own Python script to process each gate and flip-flop in the design and add its associated tracking logic. This new "GLIFTed" cache is re-inserted into the register-transfer level (RTL) processor design in the place of the original RTL cache. Pictorially, this can be seen in Figure 2. The input and output to the cache system include address and data lines and control signals (write-enable and memory stall signals); each such input and output is now associated with a taint bit which will be essential to testing whether or not information flows from our victim process V to our malicious process M.

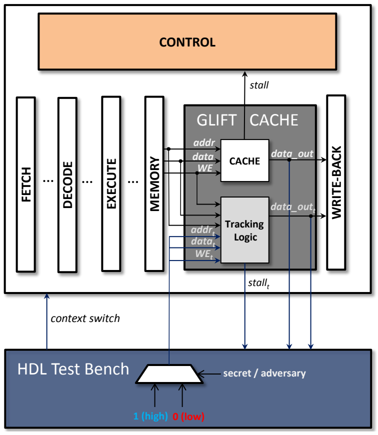

Fig. 2. A block diagram of a simple MIPS-based CPU. The cache is replaced by one which contains the original cache and its associated tracking logic, and our testbench simulates the processor to capture the output traces.

To execute the test scenario, we follow the same procedure as the previously discussed access-driven attack. We begin the simulation by having M read a fixed block of bytes D ($|D| = 256$) in a loop to fill the cache. Next, we have V execute the `aes` benchmark with all inputs to the cache marked as tainted (i.e., secret). Next, we again have M read the same data D, which models M attempting to determine the latency of its memory accesses. Since the entire system contains GLIFT logic, in simulation we are able to observe the flows of tainted (secret) information. After executing this simulation, our waveform in Modelsim indicates that tainted information does flow to M; furthermore, this information must have come from V, as there is no other possible origin. We therefore know that a flow exists, but at this stage it is still unclear whether the flow is functional or temporal.

To find exactly which type of channel was identified by GLIFT, we leverage the benefits of our model by working to identify a functional flow; as previously discussed, if we detect no functional flow, then we know the flow must be from a timing channel. To fit our model (following Figure 1), we abstracted the output of the cache as $y = \langle data\_out_M \rangle$ to indicate the cache output observable by M. Following our model, we then defined two traces: $A_1(X, k) := \langle V \text{ using } K_1 \rangle$ and $A_2(X, k) := \langle V \text{ using } K_2 \rangle$; i.e., the cases in which V encrypts using two different keys. We then simulated both of these scenarios and logged all of the discrete events of y captured by ModelSim using its discrete event list feature to obtain the output traces A_{C1} and A_{C2}. Once we collected these traces, we checked whether or not a functional flow exists for these particular traces by using a script to compare the events in each respective file. We have our script indicate a functional flow with regards to Definition 6; if there exists events $e_j \in d(A_{C1})$ and $e'_j \in d(A_{C2})$ such that $\mathsf{val}(e_j) \neq \mathsf{val}(e'_j)$. For these particular traces, our script indicates the absence of a functional flow. Since GLIFT identified a flow, we know that this must have occurred through a timing channel. Again, although the fact that no functional flow exists with respect to these particular traces does not imply the lack of a functional flow for any traces, it does lend evidence to the theory that the flow must be timing-based rather than functional (and additional testing with different keys would provide further support). Further, the implications of the existence of only a timing channel greatly depends on the threat model of the system. In many cases this issue might not be of a concern, but by concretely showing the designer that a flow is in fact from timing he can make an informed decision about its security.

IV. Conclusions and Future Work

In this work, we presented a framework that can be used to effectively separate timing flows from functional flows. Much future work is possible for both information flow tracking and for our framework in particular. First, the approach presented here focuses only on two specific traces, covering all input traces is an avenue for future work. Secondly, if a functional flow exists then we cannot say anything about the existence of a timing flow; one natural question to ask is therefore if we can identify timing channels even in the presence of a functional flow. This would have implications for applications such as data encryption, in which the output ciphertext is always a function of the secret key, yet it is critical that an adversary observing encryption not be able to deduce the secret key using a timing channel. At the present, solving such a problem seems non-trivial and we leave it as an important open problem.

References

[1] M. Tiwari, H. Wassen, B. Mazloom, S. Mysore, F. Chong, and T. Sherwood, "Complete information flow tracking from the gates up," in *Proceedings of ASPLOS 2009*, 2009.

[2] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: the case of aes," in *Proceedings of the 2006 The Cryptographers' Track at the RSA conference on Topics in Cryptology*, pp. 1–20, 2006.

[3] R. A. Kemmerer, "Shared resource matrix methodology: an approach to identifying storage and timing channels," *ACM Trans. Comput. Syst.*, pp. 256–277, 1983.

[4] W.-M. Hu, "Reducing timing channels with fuzzy time," in *Proceedings of the 1991 IEEE Symposium on Security and Privacy*, pp. 8 –20, 1991.

[5] X. Li, M. Tiwari, J. K. Oberg, V. Kashyap, F. T. Chong, T. Sherwood, and B. Hardekopf, "Caisson: a hardware description language for secure information flow," in *PLDI 2011*, pp. 109–120, 2011.

[6] T. K. Tolstrup, *Language-based Security for VHDL.* PhD thesis, Informatics and Mathematical Modelling, Technical University of Denmark, DTU, 2007.

[7] J. Oberg, W. Hu, A. Irturk, M. Tiwari, T. Sherwood, and R. Kastner, "Information flow isolation in I2C and USB," in *Proceedings of Design Automation Conference (DAC) 2011*, pp. 254 –259, 2011.

[8] E. A. Lee and A. Sangiovanni-Vincentelli, "A framework for comparing models of computation," *IEEE Transactions on Computer-Aided Design of Circuits and Systems*, vol. 17, no. 12, pp. 1217–1229, 1998.